Terms used **extensible macro language** Found **4,336** of **200,192**

Sort results by relevance
Display results expanded form
Save results to a Binder
Search Tips
Open results in a new window

Try an Advanced Search
Try this search in The ACM Guide

Results 1 - 20 of 200 Result page: **1** 2 3 4 5 6 7 8 9 10 next
Best 200 shown Relevance scale

1 Macro instruction extensions of compiler languages
M. Douglas McIlroy
April 1960 **Communications of the ACM**, Volume 3 Issue 4
Publisher: ACM Press
Full text available: pdf(831.69 KB) Additional Information: full citation, abstract, references, citings

Macroinstruction compilers constructed from a small set of functions can be made extremely powerful. In particular, conditional assembly, nested definitions, and parenthetical notation serve to make a compiler capable of accepting very general extensions to its ground language.

2 Macro processing in high-level languages
Alexander Sakharov
November 1992 **ACM SIGPLAN Notices**, Volume 27 Issue 11
Publisher: ACM Press
Full text available: pdf(709.71 KB) Additional Information: full citation, abstract, index terms

A macro language is proposed. It enables macro processing in high-level programming languages. Macro definitions in this language refer to the grammars of the respective programming languages. These macros introduce new constructs in programming languages. It is described how to automatically generate macro processors from macro definitions and programming language grammars written in the lex-yacc format. Examples of extending high-level languages by means of macros are given.

3 A model of extensible language systems
M. G. Notley
September 1971 **ACM SIGPLAN Notices , Proceedings of the international symposium on Extensible languages**, Volume 6 Issue 12
Publisher: ACM Press
Full text available: pdf(333.35 KB) Additional Information: full citation, abstract, citings, index terms

At the present time the subject of extensible languages appears to suffer from the lack of any central coherent framework to knit together the many pieces of individual work that are being done. This paper is an attempt, therefore, to fill that lack. What appears to be required is some central conceptual model or paradigm of languages and language extensibility onto which framework we can hang all these pieces of work. This paper, therefore, contains of four main point ...

4 Growing languages with metamorphic syntax macros

Claus Brabrand, Michael I. Schwartzbach

January 2002 **ACM SIGPLAN Notices , Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation PEPM '02**, Volume 37 Issue 3

Publisher: ACM Press

Full text available: pdf(217.81 KB) Additional Information: full citation, abstract, references, citings

*"From now on, a main goal in designing a language should be to plan for growth." Guy Steele: Growing a Language, OOPSLA '98 invited talk.*We present our experiences with a syntax macro language which we claim forms a general abstraction mechanism for growing (domain-specific) extensions of programming languages. Our syntax macro language is designed to guarantee *type safety* and *termination*.A concept of *metamorphisms* allows the arguments of a macro to be inductively def ...

5 Maya: multiple-dispatch syntax extension in Java

Jason Baker, Wilson C. Hsieh

May 2002 **ACM SIGPLAN Notices , Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation PLDI '02**, Volume 37 Issue 5

Publisher: ACM Press

Full text available: pdf(152.75 KB) Additional Information: full citation, abstract, references, citings, index terms

We have designed and implemented Maya, a version of Java that allows programmers to extend and reinterpret its syntax. Maya generalizes macro systems by treating grammar productions as generic functions, and semantic actions on productions as multimethods on the corresponding generic functions. Programmers can write new generic functions (i.e., grammar productions) and new multimethods (i.e., semantic actions), through which they can extend the grammar of the language and change the semantics of ...

Keywords: Java, generative programming, macros, metaprogramming

6 Composable and compilable macros:: you want it when?

Matthew Flatt

September 2002 **ACM SIGPLAN Notices , Proceedings of the seventh ACM SIGPLAN international conference on Functional programming ICFP '02**, Volume 37 Issue 9

Publisher: ACM Press

Full text available: pdf(162.46 KB) Additional Information: full citation, abstract, references, citings, index terms

Many macro systems, especially for Lisp and Scheme, allow macro transformers to perform general computation. Moreover, the language for implementing compile-time macro transformers is usually the same as the language for implementing run-time functions. As a side effect of this sharing, implementations tend to allow the mingling of compile-time values and run-time values, as well as values from separate compilations. Such mingling breaks programming tools that must parse code without executing i ...

Keywords: language tower, macros, modules

7 A brief look at extension programming before and now

Liisa Räihä

February 1995 **ACM SIGPLAN Notices**, Volume 30 Issue 2

Publisher: ACM Press

Full text available: pdf(898.76 KB) Additional Information: full citation, abstract, index terms

We try to bind together some old and some new: what is an extension. In addition, we give a short analysis of extension facilities in three language systems with slightly different theoretical basis. We compare Ada 9x, Oberon and Macro Language, with additional comments to e.g., C++.

8 Hygienic macro expansion

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, Bruce Duba
August 1986 **Proceedings of the 1986 ACM conference on LISP and functional programming LFP '86**
Publisher: ACM Press
Full text available: pdf(762.23 KB) Additional Information: full citation, references, citings

9 Staging: Assimilating MetaBorg:: embedding language tools in languages

Jonathan Riehl
October 2006 **Proceedings of the 5th international conference on Generative programming and component engineering GPCE '06**
Publisher: ACM Press
Full text available: pdf(174.54 KB) Additional Information: full citation, abstract, references, index terms

The MetaBorg usage pattern allows concrete syntax to be associated with application programmer interfaces (API's). Once a concrete syntax is defined, library writers use the Stratego language to write transformations from the concrete syntax to API data and calls in the host language. The result is a compile time translator from the combined host and domain languages to the host language. This translator is not programmable at compile time, and little or none of the infrastructure can be leverag ...

Keywords: MetaBorg, SDF, concrete syntax macros, extensible syntax, self application, staged multi--language programming, stratego

10 A language independent macro processor

William M. Waite
July 1967 **Communications of the ACM**, Volume 10 Issue 7
Publisher: ACM Press
Full text available: pdf(1.06 MB) Additional Information: full citation, abstract, references, citings, index terms

The problem of obtaining starting values for the Newton-Raphson calculation of √x on a digital computer is considered. It is shown that the conventionally used best uniform approximations to √x do not provide optimal starting values. The problem of obtaining optimal starting values is stated, and several basic results are proved. A table of optimal polynomial starting values is given.

11 Experience with an extensible language

Edgar T. Irons
January 1970 **Communications of the ACM**, Volume 13 Issue 1
Publisher: ACM Press
Full text available: pdf(1.17 MB) Additional Information: full citation, abstract, references, citings

An operational extensible language system is described. The system and its base language are appraised with respect to efficiency, flexibility, and utility for different categories of users.

Keywords: ambiguity, bootstrapping, compiler, extensible, programming languages

12 Technical contributions: Experience with extensible, portable Fortran extensions

A. James Cook
September 1976 **ACM SIGPLAN Notices**, Volume 11 Issue 9
Publisher: ACM Press
Full text available: pdf(497.43 KB) Additional Information: full citation, abstract, references, citings

We assess the impact over a three-year period, of the macro-pre-processor MORTRAN, and one of the languages it processes. We confine our assessment to SLAC and Stanford since, although MORTRAN has been widely distributed in the United States and to a lesser extent in Europe, we have no personal knowledge of its impact elsewhere. The impact is attributed to three factors: (1) portability, (2) compatibility (with existing FORTRAN libraries), and (3) extensibility, which is sub-divided into (a) ext ...

13 Technical contributions: STRCMACS: an extensive set of macros to aid in structured programming in 360/370 assembly language

C. Wrandle Barth
August 1976 **ACM SIGPLAN Notices**, Volume 11 Issue 8
Publisher: ACM Press
Full text available: pdf(219.18 KB) Additional Information: full citation, citings

14 Programmable syntax macros

Daniel Weise, Roger Crew
June 1993 **ACM SIGPLAN Notices , Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation PLDI '93**, Volume 28 Issue 6
Publisher: ACM Press
Full text available: pdf(1.09 MB) Additional Information: full citation, abstract, references, citings, index terms

Lisp has shown that a programmable syntax macro system acts as an adjunct to the compiler that gives the programmer important and powerful abstraction facilities not provided by the language. Unlike simple token substitution macros, such as are provided by CPP (the C preprocessor), syntax macros operate on Abstract Syntax Trees (ASTs). Programmable syntax macro systems have not yet been developed for syntactically rich languages such as C because rich concrete syntax requires the manual con ...

15 Tools: Expressive programs through presentation extension

Andrew D. Eisenberg, Gregor Kiczales
March 2007 **Proceedings of the 6th international conference on Aspect-oriented software development AOSD '07**
Publisher: ACM Press
Full text available: pdf(332.31 KB) Additional Information: full citation, abstract, references, index terms

Most approaches to programming language extensibility have worked by pairing syntactic extension with semantic extension. We present an approach that works through a combination of presentation extension and semantic extension. We also present an architecture for this approach, an Eclipse-based implementation targeting the Java programming language, and examples that show how presentation extension, both with and without semantic extension, can make programs more expressive.

Keywords: MOP, annotations, expressiveness, metadata, metaobject protocol

16 When and how to develop domain-specific languages

Marjan Mernik, Jan Heering, Anthony M. Sloane
December 2005 **ACM Computing Surveys (CSUR)**, Volume 37 Issue 4
Publisher: ACM Press

Full text available: pdf(318.02 KB) Additional Information: full citation, abstract, references, citings, index terms

Domain-specific languages (DSLs) are languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages in their domain of application. DSL development is hard, requiring both domain knowledge and language development expertise. Few people have both. Not surprisingly, the decision to develop a DSL is often postponed indefinitely, if considered at all, and most DSLs never get beyond the applicatio ...

Keywords: Domain-specific language, application language, domain analysis, language development system

17 MACRO: a programming language

Stephen R. Greenwood
December 1979 **ACM SIGPLAN Notices**, Volume 14 Issue 12
Publisher: ACM Press
Full text available: pdf(1.41 MB) Additional Information: full citation, references, citings, index terms

18 The theory of parsing, translation, and compiling

Alfred V. Aho, Jeffrey D. Ullman
January 1972 Book
Publisher: Prentice-Hall, Inc.

Full text available: pdf(98.28 MB) Additional Information: full citation, abstract, references, citings, index terms

From volume 1 Preface (See Front Matter for full Preface)

This book is intended for a one or two semester course in compiling theory at the senior or graduate level. It is a theoretically oriented treatment of a practical subject. Our motivation for making it so is threefold.

(1) In an area as rapidly changing as Computer Science, sound pedagogy demands that courses emphasize ideas, rather than implementation details. It is our hope that the algorithms and concepts presen ...

19 A survey of the systematic use of macros in systems building

David J. Farber
October 1971 **ACM SIGPLAN Notices , Proceedings of the SIGPLAN symposium on Languages for system implementation**, Volume 6 Issue 9
Publisher: ACM Press

Full text available: pdf(494.89 KB) Additional Information: full citation, abstract, references, citings, index terms

Assemblers with macro capabilities have been available for over ten years. There has been a limited number of mainly unpublished systematic uses of such capabilities in the construction of a variety of systems. This paper will cover a number of these cases. We will examine the features of the macro systems which allowed their usage as well as the

method used in the system implementation. We will comment on the effect on efficiency and flexibility that the use of this facility has produced.< ...

20 Pointcuts and advice in higher-order languages

David B. Tucker, Shriram Krishnamurthi
March 2003 **Proceedings of the 2nd international conference on Aspect-oriented software development AOSD '03**
Publisher: ACM Press

Full text available: pdf(987.37 KB) Additional Information: full citation, abstract, references, citings, index terms

Aspect-oriented software design will need to support languages with first-class and higher-order procedures, such as Python, Perl, ML and Scheme. These language features present both challenges and benefits for aspects. On the one hand, they force the designer to carefully address issues of scope that do not arise in first-order languages. On the other hand, these distinctions of scope make it possible to define a much richer variety of policies than first-order aspect languages permit.In this p ...

Results 1 - 20 of 200 Result page: **1** 2 3 4 5 6 7 8 9 10 next

Macro processing in high-level languages

Full text Pdf (710 KB)

Source **ACM SIGPLAN Notices** archive
Volume 27 , Issue 11 (November 1992) table of contents
Pages: 59 - 66
Year of Publication: 1992
ISSN:0362-1340

Author Alexander Sakharov

Publisher ACM Press New York, NY, USA

↑ ABSTRACT

A macro language is proposed. It enables macro processing in high-level programming languages. Macro definitions in this language refer to the grammars of the respective programming languages. These macros introduce new constructs in programming languages. It is described how to automatically generate macro processors from macro definitions and programming language grammars written in the lex-yacc format. Examples of extending high-level languages by means of macros are given.

↑ INDEX TERMS

Primary Classification:
D. Software
↳ **D.3** PROGRAMMING LANGUAGES
↳ **D.3.2** Language Classifications
↳ **Subjects:** Macro and assembly languages

General Terms:
Design, Languages, Theory

Macro instruction extensions of compiler languages

Full text Pdf (832 KB)

Source **Communications of the ACM** archive
Volume 3 , Issue 4 (April 1960) table of contents
Pages: 214 - 220
Year of Publication: 1960
ISSN:0001-0782

Author M. Douglas McIlroy Bell Telephone Lab, Inc., Murray Hill, NJ

Publisher ACM Press New York, NY, USA

↑ ABSTRACT

Macroinstruction compilers constructed from a small set of functions can be made extremely powerful. In particular, conditional assembly, nested definitions, and parenthetical notation serve to make a compiler capable of accepting very general extensions to its ground language.

↑ REFERENCES

Note: OCR errors may be found in this Reference List extracted from the full text article. ACM has opted to expose the complete List rather than only correct and linked references.

1 M. BARNET, Macro-directive approach to high speed computing. Solid State Physics Research Group, MIT, Cambridge, Mass., 1959.

2 H. B. CURRY AND R. FEYS, Combining Logic, vol, I, North Holland Publishing Co., Amsterdam, 1958, pp. 62-67.

3 D. E. EASTWOOD AND M. D. MCILROY, Macro compiler modification of SAP. Bell Telephone Laboratories Computation Center, 1959.

4 Irwin D. Greenwald, A technique for handling macro instructions, Communications of the ACM, v.2 n.11, p.21-22, Nov. 1959

5 M. HAIGH, Users specification for MICA. SHARE User's Organization for IBM 709 Electronic Data Processing Machine SHARE Secretary Distribution SSD-61, C-1462 (1959), pp. 16-63.

6 A. J. Perlis , K. Samelson, Preliminary report: international algebraic language, Communications of the ACM, v.1 n.12, p.8-22, Dec. 1958

7 A. J. PERLIS Quaterly report of the Computation Center, Carnegie Institute of Technology, Oct. 1959.

8 REMINGTON-RAND UNIVAC DIVISION, Univac generalized programming. Philadelphia, 1957.

↑ CITED BY *23*

A. Evans, Jr. , A. J. Perlis , H. Van Zoeren, The use of threaded lists in constructing a combined ALGOL and machine-like assembly processor, Communications of the ACM, v.4 n.1, p.36-41, Jan. 1961

William M. Waite, A language independent macro processor, Communications of the ACM, v.10 n.7, p.433-440, July 1967

David J. Farber, A survey of the systematic use of macros in systems building, ACM SIGPLAN Notices, v.6 n.9, p.29-36, October 1971

A. James Cook, Experience with extensible, portable Fortran extensions, ACM SIGPLAN Notices, v.11 n.9, September 1976

Thomas A. Standish, Extensibility in programming language design, ACM SIGPLAN Notices, v.10 n.7, July 1975

B. M. Leavenworth, Syntax macros and extended translation, Communications of the ACM, v.9 n.11, p.790-793, Nov. 1966

R. W. Floyd , C. N. Mooers , L. P. Deutsch, Programming languages for non-numeric processing—1: TRAC, a text handling language, Proceedings of the 1965 20th national conference, p.229-246, August 24-26, 1965, Cleveland, Ohio, United States

Eugene Kohlbecker , Daniel P. Friedman , Matthias Felleisen , Bruce Duba, Hygienic macro expansion, Proceedings of the 1986 ACM conference on LISP and functional programming, p.151-161, August 1986, Cambridge, Massachusetts, United States

James E. Emery, Small-scale software components, ACM SIGSOFT Software Engineering Notes, v.4 n.4, p.18-21, October 1979

Mark I. Halpern, Programming Languages: Toward a general processor for programming languages, Communications of the ACM, v.11 n.1, p.15-25, Jan. 1968

P. C. Poole , W. M. Waite, Machine independent software, Proceedings of the second symposium on Operating systems principles, October 20-22, 1969, Princeton, New Jersey

N. J. Elias , A. W. Wetzel, The IC Module Compiler, a VLSI system design aid, Proceedings of the 20th conference on Design automation, p.46-49, June 27-29, 1983, Miami Beach, Florida, United States

Joe M. Thames, Jr., SLANG a problem solving language for continuous-model simulation and optimization, Proceedings of the 1969 24th national conference, p.23-41, August 26-28, 1969

Dennis M. Ritchie, The development of the C language, ACM SIGPLAN Notices, v.28 n.3, p.201-208,

March 1993

Dennis M. Ritchie, The development of the C programming language, History of programming languages---II, ACM Press, New York, NY, 1996

Eric Van Wyk, Specification languages in algebraic compilers, Theoretical Computer Science, v.291 n.3, p.351-385, 6 January 2003

Douglas T. Ross, Origins of the APT language for automatically programmed tools, ACM SIGPLAN Notices, v.13 n.8, p.61-99, August 1978

Douglas T. Ross, Origins of the APT language for automatically programmed tools, History of programming languages I, ACM Press, New York, NY, 1978

David Beech, A Structural View of PL/I, ACM Computing Surveys (CSUR), v.2 n.1, p.33-64, March 1970

David A. Fisher, A survey of control structures in programming languages, ACM SIGPLAN Notices, v.7 n.11, November 1972

Mathew N. Matelan, MPACT: microprocessor application to control-firmware translator, ACM SIGDA Newsletter, v.5 n.1, p.13-41, March 1975

John R. Metzner, A graded bibliography on macro systems and extensible languages, ACM SIGPLAN Notices, v.14 n.1, p.57-64, January 1979

Jerome Feldman , David Gries, Translator writing systems, Communications of the ACM, v.11 n.2, p.77-113, Feb. 1968

↑ **Collaborative Colleagues:**

M. Douglas McIlroy: Jon L. Bentley

↑ **Peer to Peer - Readers of this Article have also read:**

- Data structures for quadtree approximation and compression **Communications of the ACM** 28, 9
  Hanan Samet
- A hierarchical single-key-lock access control using the Chinese remainder theorem **Proceedings of the 1992 ACM/SIGAPP Symposium on Applied computing**
  Kim S. Lee , Huizhu Lu , D. D. Fisher
- The GemStone object database management system **Communications of the ACM** 34, 10
  Paul Butterworth , Allen Otis , Jacob Stein
- Putting innovation to work: adoption strategies for multimedia communication systems **Communications of the ACM** 34, 12
  Ellen Francik , Susan Ehrlich Rudman , Donna Cooper , Stephen Levine
- An intelligent component database for behavioral synthesis **Proceedings of the 27th ACM/IEEE conference on Design automation**
  Gwo-Dong Chen , Daniel D. Gajski

Web Results **141** - **150** of about **214,000** for **extensible macro language >1997**. (**0.13** seconds)

[PDF] Designing **Extensible** IP Router Software
File Format: PDF/Adobe Acrobat - View as HTML
within such a **macro**-stage. XORP thus further decom- ... XRLs to be called from any scripting **language** via a sim- ... 7007 explanation and apology, **1997**. ...
www.xorp.org/papers/xorp-nsdi.pdf - Similar pages

[PDF] Little Languages and their Programming Environments
File Format: PDF/Adobe Acrobat - View as HTML
XML (for "**eXtensible** Markup **Language**") is a proposed standard for a ... Instead, Scheme may implement let with a **macro** that elaborates each use of the form ...
www.brinckerhoff.org/JBCsite/papers/mw01-cgkf.pdf - Similar pages

[PDF] Static Analysis for Syntax Objects
File Format: PDF/Adobe Acrobat - View as HTML
guages]: **Language** Classifications—**extensible** languages, **macro** ... nipulation, pages 203–217, New York, NY, USA, **1997**. ACM Press. ...
www-static.cc.gatech.edu/~shivers/papers/ziggurat.pdf - Similar pages

Languages for the Java VM
Jickle is similar to a **macro language** for applications. ... **Extensible** semantics is a distinguishing feature of Lua. ...
flp.cs.tu-berlin.de/~tolk/vm**languages**.html - 102k - Cached - Similar pages

[PDF] Abstract 1 Introduction 2 Outline of **Extensible** Java Pre-processor EPP
File Format: PDF/Adobe Acrobat - View as HTML
for **language** researchers; a framework for **extensible**. Java implementation; and a framework for a Java ... parsing, **macro** expansion of extended syntax, and ...
staff.aist.go.jp/y-ichisugi/epp/edoc/epp-parser.pdf - Similar pages

[Paper] An **Extensible** Data Mining and Pattern Recognition System
Inside integration models, of the architectures that handle several tools defined in [4], the **extensible** system proposed works in the **Macro** level. ...
www.actapress.com/PDFViewer.aspx?paperId=14031 - Similar pages

JOT: Journal of Object Technology - BON-CASE: An **Extensible** CASE ...
We describe BON-CASE, an **extensible** tool for the BON modelling **language**. ... One example of this is the U2B **macro** package [24], which translates UML models ...
www.jot.fm/issues/issue_2002_08/article5 - 63k - Cached - Similar pages

[PDF] Hey, You Got Your **Language** In My Operating System!
File Format: PDF/Adobe Acrobat - View as HTML
The drive toward **extensible** operating systems can. be viewed as trying to make the operating system as. flexible as a **language**/library environment. Features ...
ftp://ftp.cs.dartmouth.edu/TR/TR98-340.pdf - Similar pages

WWW-Talk Apr-Jun 1994: Re: Concerns about HTML+ complexity (example)
It's extremely **extensible**. Each visual element (possibly each geometry ... There should be a simple **macro language** to do the conversion: ...
1997.webhistory.org/www.lists/www-talk.1994q2/1059.html - 8k - Cached - Similar pages

Introduction to the Unix shell history
Tcl was first scripting **language** designed to be a **macro language** for other ... Like Tcl, ksh93 is **extensible** and embeddable with a C **language** API. ...
www.softpanorama.org/People/Shell_giants/introduction.shtml - 22k -
Cached - Similar pages

Result Page: **Previous** 5 6 7 8 9 10 11 12 13 14 **15** 16 17 18 19 20 21 22 23 24 **Next**

---

extensible macro language >1997 Search

Search within results | Language Tools | Search Tips

A model of extensible language systems

Full text Pdf (333 KB)

Source **Proceedings of the international symposium on Extensible languages** table of contents
Grenoble, France
Pages: 29 - 38
Year of Publication: 1971
Also published in ...

Author M. G. Notley Scientific Centre, IBM (UK) Ltd, Neville Road, Peterlee, Co Durham

Sponsor SIGPLAN: ACM Special Interest Group on Programming Languages

Publisher ACM Press New York, NY, USA

DOI Bookmark: Use this link to bookmark this Article: http://doi.acm.org/10.1145/800006.807977
What is a DOI?

↑ ABSTRACT

At the present time the subject of extensible languages appears to suffer from the lack of any central coherent framework to knit together the many pieces of individual work that are being done. This paper is an attempt, therefore, to fill that lack. What appears to be required is some central conceptual model or paradigm of languages and language extensibility onto which framework we can hang all these pieces of work. This paper, therefore, contains of four main points:- * The description of a very simple model of formal machine languages and their operation. * A discussion of the generality of this model. * A discussion of the usefulness of this model for the study of extensible languages. * An illustration of the application of the model by reference to an on-going implementation at the IBM (UK) Scientific Centre.

↑ CITED BY

John R. Metzner, A graded bibliography on macro systems and extensible languages, ACM SIGPLAN Notices, v.14 n.1, p.57-64, January 1979

↑ INDEX TERMS

Primary Classification:
D. Software
↳ **D.3** PROGRAMMING LANGUAGES
↳ **D.3.2** Language Classifications

Subjects: Extensible languages

General Terms:
Languages

This Article has also been published in:

- **ACM SIGPLAN Notices**
  Volume 6 , Issue 12 December 1971

3. String expressions (S)

S ~ (S)
- ~ S(n, ααα)
- ~ F( ), where F is any function defining a string value
- ~ ST
- ~ $ST_1 \oplus ST_2$ Concatenate string ST_2 to end of string ST_1
- ~ $ST_1 \ominus ST_2$ Remove all ST_2 strings from string ST_1
- ~ ST ⊕ SY Add symbol SY to end of string ST
- ~ SY ⊕ ST Add symbol SY to beginning of string ST

4. String Functions

Representation	Name	Class of Value Defined	Description of Value
Norm (ST)	Norm	I	Positive integer equal to the number of symbols in the string ST.
GETNS (ST, I)	Get nth symbol	SY	I-th symbol of string ST
FIRST (ST)	First symbol of string	SY	GETNS (ST, 1)
SUB (ST, I1, I2)	SUBSTRING	ST	~ as defined in [1]
FOS (ST, SY)	First occurrence of symbol	I	If the symbol SY is in the string ST, then the value defined is an integer equal to the position which the first SY occupies; otherwise the value is zero.
AI (I)	Transform an integer to a string	ST	
EI (ST)	Transform a string to an integer	I	

5. Procedures

Call	Name	Description
FREES (ST)	Free string	ST := NULL STRING
REMNS (ST, I, SY)	Remove I-th symbol	Set SY equal to the I-th symbol of string, Remove I-th symbol from string ST.
POINT (ST, I_1, I_2)	Set pointer	Set integer I_2 to designate the I-th symbol of string ST.
SEQ (I_2, SY, L)	Sequence	Set symbol SY equal to the symbol which I_2 designates and advance I_2 to the next symbol in the string. If the procedure is called after I_2 has passed the last symbol of the string, transfer to statement L.
INSRT (ST, I, SY)	INSERT	INSERT the symbol SY between the I-th and the (I + 1)-th symbols of the string ST so that SY becomes the (I+1)-th symbol of string.
REPLS (ST, SY_1, SY_2)	Replace symbol by symbol	Wherever the symbol SY_1 occurs in string ST, replace it by the symbol SY_2.
RPLST (ST_1, SY, ST_2)	Replace symbol by string	Wherever the symbol SY_1 occurs in string ST_1, replace it by the string ST_2.

Reference

1. Julien Green et al. Remarks on Algol and symbol manipulation, *Comm. Assoc. Comp. Mach. 2*, No. 9 (Sept. 1959).

Macro Instruction Extensions of Compiler Languages

M. Douglas McIlroy, *Bell Telephone Laboratories, Inc., Murray Hill, New Jersey*

Abstract. Macroinstruction compilers constructed from a small set of functions can be made extremely powerful. In particular, conditional assembly, nested definitions, and parenthetical notation serve to make a compiler capable of accepting very general extensions to its ground language.

1. Informal Development

The idea of macroinstructions is not new. Many existing compilers permit macros in forms more or less sophisticated. Complex macro operations, however, are often quite laborious to incorporate in a programming system because special purpose generators must be built to handle each addition. It is our aim to show a limited set of functions readily implemented for a wide variety of programming systems which constitute a powerful tool for extending source languages conveniently and at will. Our development will be for the most part informal, avoiding technical detail. We will illustrate ideas with an informal compiler as we go along leaving one formal realization based on Algol for the appendix. As a source of examples, we have chosen the familiar field of algebraic translation.

At base, a macroinstruction is simply a pattern for an open subroutine. For example, a single-address machine might be converted in appearance to a three-address machine by defining a set of macros such as

$$\begin{aligned} \text{ADD, A, B, C} \equiv\ & \text{FETCH, A} \\ & \text{ADD, B} \\ & \text{STORE, C} \end{aligned} \tag{1}$$

In (1) the identity sign, $\equiv$, is used to separate the definiendum on the left from the definiens on the right. The first component—ADD—on the left names the macro being defined as a function of the dummy parameters specified by the remaining components, A, B and C. A compiler designed to accept macro definitions could read (1) and therefore properly assemble any macro call such as

$$\text{ADD, X, Y, Z} \tag{2}$$

1.1. *Pyramided Definitions.* An obvious extension of this simple macro compiler is to allow new definitions in terms of old as in

$$\begin{aligned} \text{COMPLEXADD, A, B, C} \equiv\ & \text{ADD, A, B, C} \\ & \text{ADD, A + 1, B + 1, C + 1} \end{aligned}$$

which defines three-address addition of complex numbers whose real and imaginary parts are stored sequentially in terms of real three-address addition.

1.2. *Conditional Macros.* Until recently, macro compilers did not grow beyond this level of sophistication, but such a compiler is weak and inflexible. For example, a programmer using the three-address addition of (1) would be most unhappy if he had to store a result in C that was intended to be used only in the very next instruction. In order to add P + Q + R and store the result in S, he would like to make a call such as

$$\begin{aligned} & \text{ADD, P, Q, ACC} \\ & \text{ADD, ACC, R, S} \end{aligned} \tag{3}$$

where ACC is used to refer to the accumulator as a special address. The compiler should recognize this special symbol and construct code appropriately. Such conditional assembly might be specified in the definition

$$\begin{aligned} \text{ADD, A, B, C} \equiv\ & \begin{cases} \text{if A is not ACC} \\ \text{FETCH, A} \end{cases} \\ & \text{ADD, B} \\ & \begin{cases} \text{if C is not ACC} \\ \text{STORE, C} \end{cases} \end{aligned} \tag{4}$$

Given this definition, the compiler would produce efficient code from the call (3), purged of redundant STORE and FETCH instructions:

$$\begin{aligned} & \text{FETCH, P} \\ & \text{ADD, Q} \\ & \text{ADD, R} \\ & \text{STORE, S.} \end{aligned}$$

For an actual machine-language compiler, one would devise a standard shorthand notation for the conditions in (4), but that is inessential to this discussion.

1.3. *Created Symbols.* The matter of labels on coding lines within a macro definition proves bothersome in cases such as

$$\begin{aligned} \text{OVERDRAW, X, Y, ACTION, Z} \equiv\ & \text{FETCH, X} \\ & \text{SUB, Y} \\ & \text{PLUSJUMP, Z} \\ & \text{ACTION} \\ & \text{Z: empty line of code} \end{aligned}$$

Here the label Z must be supplied as a parameter in order to assure that it gets a unique name each time OVERDRAW is called. But, since the label has no significance outside the macro, its naming should not be of concern to the programmer. This problem may be handled by providing a mechanism to create names, for example

$$\begin{aligned} \text{OVERDRAW, X, Y, ACTION} \equiv\ & \text{creat Z} \\ & \text{FETCH, X} \\ & \text{SUB, Y} \\ & \text{PLUSJUMP, Z} \\ & \text{ACTION} \\ & \text{Z: empty line of code} \end{aligned} \tag{5}$$

Created names will be generated sequentially as needed from some special alphabet that the programmer is forbidden to use.

1.4. *Grouping by Parentheses.* The example (5) illustrates another difficulty in our simple compiler: Suppose the programmer wishes to put SETZERO, X in place of ACTION. In order to avoid ambiguity in the call, some indication of grouping is required. Parentheses are used to accomplish this:

$$\text{OVERDRAW, X, Y, (SETZERO, X)}$$

Parentheses so used result in standard notation for compounding functions.

A simple algebraic translator can be built from the macro compiler as already described. For example one might define

$$\begin{aligned} \text{A} \equiv\ & \text{FETCH, A} \\ \text{B} \equiv\ & \text{FETCH, B} \\ & \text{etc.} \end{aligned} \tag{6}$$

$$\begin{aligned} \text{SET, X, Y} \equiv\ & \text{Y} \\ & \text{STORE, X} \\ \text{SUM, X, Y} \equiv\ & \text{X} \\ & \text{STORE, T} \\ & \text{T: redefine to be T + 1} \\ & \text{Y} \\ & \text{T: redefine to be T − 1} \\ & \text{ADD, T} \\ \text{COS, X} \equiv\ & \text{X} \\ & \text{SUBJUMP, COS} \end{aligned}$$

The machine is assumed to have a "subroutine jump" operation, SUBJUMP, the compiler is assumed to have a pseudo-operation "redefine" that allows one to modify equivalences of symbols during compilation. A block of storage must have been set aside at location T for temporary storage of intermediate results. Repetitive redefini-

tion of T accomplishes overlapping but noninterfering use of this temporary storage.

With the definitions (6) one may write suggestive calls in functional notation, for example

$$\text{SET, C, (SUM, A (COS, (SUM, A, B)))} \qquad (7)$$

means in ALGOL notation

$$C := A + \cos (A + B)$$

The coding produced from the call (7) by our "algebraic translator" is shown in figure 1.

The scope of each macro is indicated to the right of the code. This naive translator produces inefficient code, but one could take advantage of conditional assembly macros to improve it. Thus the macro SUM is more advantageously defined by

$$\text{SUM, X, Y} \equiv \begin{cases} \text{if Y is a symbol} \\ \quad \text{X} \\ \quad \text{ADD, Y} \\ \text{if Y is not a symbol} \\ \quad \text{X} \\ \quad \text{STORE, T} \\ \text{T: redefine T + 1} \\ \quad \text{Y} \\ \text{T: redefine T - 1} \\ \quad \text{ADD, T} \end{cases} \qquad (8)$$

Such ideas may be elaborated at will.

1.5. *Nested Definitions.* Among the definitions (6) was included for each variable a definition such as A ≡ FETCH, A. The programmer must remember to make a definition like this for every variable storage name he assigned. Since the pattern is well fixed, why not let the macro compiler do the work. Assignment of variable storage and definition of the associated macro can be done simultaneously by

$$\text{VARIABLE, A} \equiv \text{A: reserve 1} \quad \text{A} \equiv \text{FETCH, A} \qquad (9)$$

where "reserve n" is a pseudo-operation to set aside n storage locations. Buried within the definition for VARIABLE is a second definition scheme that produces a new definition as a result of every call for VARIABLE. The programmer is now insulated from the internal mechanism of the translator. However, he is still saddled with repetitive work in writing

VARIABLE, X
VARIABLE, Y
etc.

He may be relieved of this burden by adding an iterative feature to the compiler, say in the form

$$\text{VARIABLES, X} \equiv \text{repeat over X} \quad \text{VARIABLE, X} \qquad (10)$$

where "repeat over X" says that X is expected to be a list and that the macro should be expanded over and over for each entry on the list until it is exhausted. Thus

$$\text{VARIABLES (X, Y, Z)} \qquad (11)$$

is equivalent to the sequence of calls

VARIABLE, X
VARIABLE, Y
VARIABLE, Z

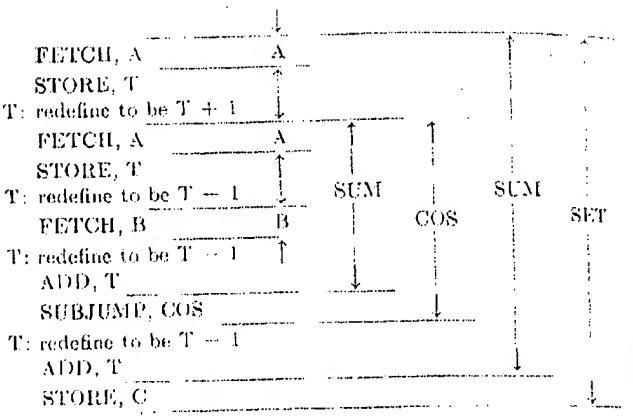

FIG. 1. A compilation of SET, C, (SUM, A, (COS, (SUM, A, B)))

2. Varying Language Styles

The new aspect of a source language to which definitions have been added may be termed a *definitional extension* of that language. This term is borrowed from logic (Curry). A macro compiler is seen to be nothing more than a machine for performing logical reductions from the extended language to the ground language. Since the rules for reduction are not part of the ground language, a macro compiler might be made to yield as output any one of a large class of ground languages. It appears that most existing machine-input languages might be subsumed under a properly designed macro compiler that could be fairly readily implemented. (As a particular example, the author has produced FORTRAN algebraic compiler language as output from a relatively naive working compiler for the SAP machine assembly language on the IBM 704.)

The principal problem in implementing a compiler of varying ground languages is making the extensions of the languages appear natural to users of each. This is mostly a question of adopting the compiler's method of scanning to accept input in ground language style. As an added dividend such a compiler will free the tight style restrictions of any one ground language, allowing great latitude in specifying the appearance of its extensions.

Scanners included in most compilers work in one of two quite simple fashions, usually called fixed and variable field scanning. The former method, gradually losing importance by comparison to variable field scanning, will be neglected in this discussion. Variable field scanning involves extracting linguistic elements as blocks of char-

acters separated by selected break characters. Break characters may serve merely as separators (e.g., commas to indicate parallel structure), as signs denoting a treatment to be performed on the separated blocks (e.g., the operation signs in ALPHA = BETA + GAMMA), or as delimiters (paired parentheses) to impose tree structure on otherwise linear text. Occasionally break "characters" may actually be blocks such as the **begin** and **end** delimiters in ALGOL, but this obvious extension is not hard to handle.

Typical operation of a scanner consists in scanning to the next break character to isolate a linguistic element and then performing some action signified by the break character or by the isolated element. High flexibility of source language style may be accomplished simply by permitting the set of break characters to be modified. Further power accrues from including directions as to the interpretation of break characters. Both improvements may be bought cheaply in a properly designed macro compiler. For example, completely parenthesized algebraic expressions on binary operators might be added to the source language of our informal compiler whose ground language is single-address symbolic machine language as follows:

First specify the break characters to be used in decoding calls for algebraic macros. This might be done by defining ALGEBRAIC, a macro that signals the compiler to modify its set of break characters

$$\text{ALGEBRAIC} \equiv \text{replace the comma break class by } , + - \times / = \tag{12}$$

then, similarly to (6), defining for each symbol a macro like

$$\begin{aligned} A = X \equiv\ & \text{ALG, (X)} \\ & \text{STORE, A} \end{aligned} \tag{13}$$

(Again, the programmer may be relieved of writing the details of such definitions by burying (13) inside another macro.) The macro ALG will write a program for computing the value of expression X. As shown in figure 2, ALG is quite lengthy, but in execution not prohibitively long because large segments are alternatives, so that the whole macro will not be executed for any one particular subexpression. The definition is recursive by virtue of the macro definition Z ≡ ALG, (X) which will cause the first operand of a binary operation to be evaluated and the macro call T = (X) for the second. We have again used T as the symbolic address of a zone of temporary storage common to all algebraic expressions. The **repeat** statement is used to perform a scan of the algebraic expression. The inner definition Z ≡ ALG, (X) is used as a way to save the first operand during the scan of the second. The quantity C, simply a parameter in the ground language, is used to determine which operand is currently being examined and to remember the break character during the scan of the second.

```
ALG, X ≡ if X is not a symbol go to 5
         FETCH, X
         go to 7
       5: create Z, C
         repeat over X thru 7
         if C is defined go to 6
       Z ≡ ALG, (X)
         if break is not + go to 8
       C: redefine to be 1
         go to 7
       8: if break is not − go to 9
       C: redefine to be 2
         go to 7
       9: ···
         [similarly set C = 3 for ×, C = 4 for /]
         ···
       6: empty line of code
         T = (X)
       T: redefine to be T + 1
         Z
       T: redefine to be T − 1
         go to C
       1: ADD, T
         go to 7
       2: SUB, T
         go to 7
       3: MULT, T
         go to 7
       4: DIV, T
         go to 7
       7: empty line of code
```

FIG. 2. A macro for algebraic translation

It may be useful to regard algebraic translation as a special case of macro expansion,—translating a formula into machine code is nothing more than recursively placing parameters into certain open subroutine patterns.

For the sake of clarity, we have not invented symbolism for the various special types of statement used in the definition. A uniform grammar conformable to that of the ground language would be easier to mechanize. It is sufficient to note that there are only the following types of statements peculiar to macros:

1. **if** . . . **go to** n
   a. **go to** n
   b. **if** X is (not) Y **go to** n
   c. **if** X is (not) of class Y
   d. **if** XRY **go to** n
      where R is >, =, <, ≠, etc.
   e. **if** break is (not) X **go to** n
2. **repeat** over X thru n
3. **create** X
4. **replace** the X break class by Y

3. Recapitulation

Having completed the informal outline of a macro compiler, let us summarize its salient features:

1. definitions may contain macro calls
2. parenthetical notation for compounding calls
3. conditional assembly
4. created symbols
5. definitions may contain definition schemata
6. repetition over a list

Items 1 and 5 are examples of a familiar propensity of mathematicians to widen the domain of applicability of any particular concept as far as possible. Applied to macros the principle becomes: *Allow anything in the body of a definition that is acceptable outside.* Items 2, 3, 4 and 6 transfer the same spirit to a higher level; all are analogs of types of statements found convenient in algebraic translators, but their effect is at compile time rather than object time. Parenthetical notation allows compounding of compiler functions just as it provided compounding of object functions in an algebraic translator. Conditional assembly is analogous to ALGOL **if** statements. Created symbols serve as internal names in much the same way as those generated by algebraic translators. Repetition over a list corresponds to **for** statements. To recapitulate: *Statements effective at object time should have counterparts effective at compile time.* Couched in yet another way: a compiler should include within it an interpreter for its source language or something equivalent.

4. Significance of Nested Definitions

Surprising power is achieved as a result of allowing the definiens of a macro instruction to include definition schemata. Trivial examples like that of (8) are easily multiplied; there nested definitions were used only to save the programmer obvious repetitive writing. Far more significant is the use in figure 2 of the hidden macro Z. In effect the macro definition here serves as a temporary storage for a subexpression while the scanner simulated by "repeat" isolates and translates the next subexpression. In like ways nested macros can be used to accomplish complex symbol manipulating tasks within the compiler. Using this feature the author programed a Bell Telephone Laboratories compiler (Eastwood) to do symbolic differentiation and to be a universal Turing machine. These demonstrations in themselves are of little value[1]; it is in performing symbol manipulation tasks incidental to new source language styles that generated macro definitions come to the fore. Conditional assembly and repetition over a list were offered in section 3 as examples of the principle that operations (in this case conditional transfer and indexing) effective at object time should have counterparts effective at compile time. Nested definitions turn out not only to be ways of generating sets of similar definitions, but also to be the counterpart of the fundamental machine operation STORE.

5. Implementation

We describe the apparatus that must be added to a compiler to convert it to handle macroinstructions. Existing scanning mechanisms can be adapted to examine macro definitions, identifying occurrences of dummy parameters, and to analyze the parameter string of a macro call. Only the simplest parenthesis analysis need be available—identifying paired outermost parentheses in macro calls; and perhaps identifying grouping of statements as was indicated by brackets in (4).

A table of definitions must be constructed, probably in a condensed notation (cf. Greenwald) with occurrences of dummy parameters replaced by uniform placeholders, say (1), (2), (3), · · · . Further if the scheme of **go to** statements indicated in figure 2 is to be used, the labels to which they refer must be located.[2]

A generator routine must be available to take the parameter list of a macro call and the definition and from these to generate line-by-line coding. The resultant code is turned over to the compiler for further processing, just as if the code had come from normal input channels. The generator must be recursive in that it can leave off expanding a macro and continue on the inner one whenever a macro call is generated, returning to the outer macro upon completing the inner expansion.

In their simplest realization, special statements such as **create** and **if** . . . **go to** . . . may be added straightforwardly to the normal complement of compiler statements. They would then be generated during macro expansion just like any other line of code. Since these statements have meaning only to the macro facility, they might instead be identified at definition time and specially flagged in order to speed the generating process.

Using the logic outlined here, it turns out that nothing special need be done in order to allow nested definitions.

7. Origins of these Techniques

Though most of this article reflects lore current throughout the computing world, priority for a few of the ideas contained here can be assigned to individuals. Earlier programing systems (Remington-Rand UNIVAC) have included extensive use of generative techniques, but typically generators were described in ground language. Macros as outlined here are in effect descriptions of such generators. Their work is done by one generator working interpretively from definition schemata. Conditional macros were devised independently by several persons besides the author within the past year. In particular, A. Perlis pointed out that algorithms for algebraic translation could be expressed in terms of conditional macros. Some uses of nested definitions were discovered by the author; their first application in symbol manipulation was by J. Bennett, also of Bell Telephone Laboratories. Repetition over lists is due to V. Vyssotsky. Perlis also noted that macro compiling may be done by routines to a large degree independent of ground language. One existing macro compiler, MICA (Haigh), though working in only one ground language is physically separated from its ground-language compiler. An analyzer of variable-style source languages exists in the SHADOW routine of M. Barnett, but lacks an associated mechanism for incorporating extensions. Created

[1] The Turing machine demonstration does prove that all machine operations have some image in a macro compiler, or in other words that such a compiler is a general purpose computer.

[2] It is sufficient to locate the labels alone and not the references. One may look up the location of labels at expansion time.

symbols and parenthetical notation are obvious loans from the well-known art of algebraic translation.

8. Appendix

As a concrete example of a macro compiler language of the type discussed in this article, we describe such a compiler built around ALGOL. Two new declarations, **macro** and **text**, are added to the already existing ALGOL set. Existing apparatus such as **if, for, begin, end** are taken over with obvious application in macros. The **for** statement has been generalized particularly for convenience in macros. The **copy** statement becomes superfluous as its work may be done by macros.

Macro *declaration.* A **macro** declaration specifies a pattern of code with replaceable identifiers to be called for by a later macro statement. Replaceable (bound) identifiers are of two sorts, ordinary and created parameters.

Form: $\Delta \sim$ **macro** I(I, I, $\cdots$, I; I, I, $\cdots$, I) $:= \pi$

The identifier before the parentheses is the macro name. Identifiers within parentheses are parameters, ordinary before the semicolon, created after.

The following alternative forms are permissible in special cases:

(i) **macro** I $:= \pi$ means
macro I(;) $:= \pi$

(ii) **macro** I(I, I, $\cdots$, I) $:= \pi$ means
macro I(I, I, $\cdots$, I;) $:= \pi$

The symbol π stands for a program consisting of one ALGOL statement or a sequence of ALGOL statements delimited by **begin** and **end** or by **begin** and **end** I, where I is the macro name. If **text** does not appear explicitly in π, its presence is assumed. Macro definitions are dynamically replaceable.

Text *Declaration.*

Form: $\Delta \sim$ **text** π

Within a macro, a **text** declaration specifies portions of program that are to replace the macro when it is called. Parts of a definition outside **text** are interpreted as instructions to the compiler.

A **macro** statement is a call for an already defined macro.

Form: I(X, X, $\cdots$, X)

The strings, X, of symbols are substituted for appearances of corresponding ordinary parameters through the definition. Outermost parentheses are stripped from each X before substitution. This substituted segment functionally replaces the macro statement in the program.

Generalized **for** *Statement.*

Form: $\Sigma \sim$ **for** I $:=$ X, X, $\cdots$, X, I $=$ X, X, $\cdots$, I $:=$ X, X, $\cdots$, X π

Let the number of X's associated with each identifier I be n. Then the program π is to be repeated n times with the occurrences of each I replaced in turn by successive X's from its associated list. (X may stand for any kind of ALGOL object, provided only that it makes sense to place it for an occurrence of I.)

The following example shows two ways of defining the ALGOL alternative statement using macros. In the first way, "alternative" will be used in a form such as

```
alternative ((a > 0, a < 0, a = 0),
             (y := a + 2,  y := a - 2,  y := 0))
```

meaning assign the values a + 2, a − 2, or 0 to y according as a is positive, negative, or zero. Alternative has two parameters, the first a list of conditions, the second a list of outcomes associated with the conditions

```
macro alternative (B, S; X, Y, Z) :=
  begin
    for X := B,  Y := S
      text if X
        begin Y;  go to Z end
    text Z;
  end alternative
```

The second way of defining alternative makes it a one-parameter macro, that parameter being a list of grouped conditions and outcomes. It would be used after the fashion

```
alternative (((a > 0, yields, y := a + 2),
              (a < 0, yields, y := a - 2),
              (a = 0, yields  y := 0)))
```

The noise word "yields" has been added for the sake of literacy. Had some scanner-modifying apparatus been included in the specifications of ALGOL, nuisances such as the commas around "yields" and the extra pair of parentheses might have been eliminated. This version of alternative is defined by a two-level declaration:

```
macro alternative (BS; X, Y)
  begin
    for X := BS
      text subaltern (X, Y)
    text Y;
  end
macro subaltern (B, yields, S, Y)
  if B begin S; go to Y end
```

A simple variant on the inner macro subaltern provides the possibility of statements like

```
alternative (((a > 0, yields, y := a + 2),
              (a < 0, yields, y := a - 2),
              (otherwise, y := 0)))
macro subaltern (B, Z, S, Y)
  begin
    if B = "otherwise"
      text Z
    if B ≠ "otherwise"
      text S
    text go to Y
  end subaltern
```

BIBLIOGRAPHY

M. Barnett, Macro-directive approach to high speed computing. Solid State Physics Research Group, MIT, Cambridge, Mass., 1959.

H. B. Curry and R. Feys, *Combinatory Logic*, vol. I. North Holland Publishing Co., Amsterdam, 1958, pp. 62-67.

D. E. Eastwood and M. D. McIlroy, Macro compiler modification of SAP. Bell Telephone Laboratories Computation Center, 1959.

I. D. Greenwald, Handling of macro instructions. *Comm. Assoc. Comp. Mach.* 2, No. 11 (1959), 21-22.

M. Haigh, Users specification for MICA. SHARE User's Organization for IBM 709 Electronic Data Processing Machine, SHARE Secretary Distribution SSD-61, C-1462 (1959), pp. 16-63.

A. J. Perlis, Official Notice on Algol language. *Comm. Assoc. Comp. Mach.* 1, No 12 (1958), 8-22.

A. J. Perlis, Quarterly report of the Computation Center, Carnegie Institute of Technology, Oct. 1959.

Remington-Rand UNIVAC Division, Univac generalized programing. Philadelphia, 1957.

Proving Theorems by Pattern Recognition I

Hao Wang, *Bell Telephone Laboratories, Murray Hill, New Jersey**

1. Introduction

Certain preliminary results on doing mathematics by machines ("mechanical mathematics") were reported in an earlier paper [20]. The writer suggested developing inferential analysis as a branch of applied logic and as a sister discipline of numerical analysis. This analogy rests on the basic distinction of pure existence proofs, elegant procedures which in theory always terminate, and efficient procedures which are more complex to describe but can more feasibly be carried out in practice. In contrast with pure logic, the chief emphasis of inferential analysis is on the efficiency of algorithms, which is usually attained by paying a great deal more attention to the detailed structures of the problems and their solutions, to take advantage of possible systematic short cuts. The possibilities of much more elaborate calculations by machines provide an incentive to studying a group of rather minute questions which were formerly regarded as of small theoretical interest. When the range of actual human computation was narrow, there seemed little point in obtaining faster procedures which were still far beyond what was feasible. Furthermore, on account of the versatility of machines, it now appears that as more progress is made, strategies in the search for proofs, or what are often called heuristic methods, will also gradually become part of the subject matter of inferential analysis. An analogous situation in numerical analysis would be, for example, to make the machine choose to apply different tricks such as taking the Fourier transform to obtain a solution of some differential equation.

The present paper is devoted to a report on further results by machines and an outline of a fairly concrete plan for carrying the work to more difficult regions. A fundamentally new feature beyond the previous paper is a suggestion to replace essentially exhaustive methods by a study of the patterns according to which extensions involved in the search for a proof (or disproof) are continued. The writer feels that the use of pattern recognition, which is in the cases relevant here quite directly mechanizable, will greatly extend the range of theorems provable by machines.

As is to be expected, the actual realization of the plan requires a large amount of detailed work in coding and its more immediate preparations. The machine program P completed so far on an IBM 704 contains only a groundwork for developing the method of pattern recognition. It already is rather impressive insofar as ordinary logic is concerned but has yet a long way to go before truly significant mathematical theorems can be proved. For example, the program P has to be extended in several basic directions before a proof can be obtained for the theorem that the square root of 2 is not a rational number. On the other hand, theorems in the logical calculus can be proved very quickly by P. There are in *Principia Mathematica* altogether over 350 theorems strictly in the domain of logic, viz., the predicate calculus with equality, falling in 9 chapters (1 to 13, since there are no 6, 7, 8, and since 12 contains no theorems). The totality of these is proved with detailed proofs printed out by the program P in about 8.4 minutes. To prove these theorems, only about half—and the easier half—of P is needed. The other half of P can prove and disprove considerably harder statements and provides at the same time groundwork for handling all inferential statements. This program P will be described in section 2.

Since the central method to be discussed is primarily concerned with the predicate calculus, its wider signifi-

* On leave of absence from University of Oxford, Oxford, England, for 1959-60.